CRT-676

SHORTEST PATH ALGORITHMS: A COMPUTATIONAL STUDY WITH THE C PROGRAMMING LANGUAGE*

by

Jean-François Mondou[1]
Teodor Gabriel Crainic[1,2]
Sang Nguyen[1]

[1] Centre de recherche sur les transports, Université de Montréal, C.P. 6128, Succursale "A", Montréal, Québec, Canada H3C 3J7.

[2] Département des sciences administratives, Université du Québec à Montréal, C.P. 6192, Succursale "A", Montréal, Québec, Canada H3C 4R2.

* This research was supported by a grant from Natural Sciences and Engineering Research Council of Canada and from F.C.A.R., Québec.

Centre de recherche sur les transports - Publication #676

December 1989

90-03005/

Dépôt légal - 1er trimestre 1990

ABSTRACT

The main purpose of this study is to evaluate the computational efficiency of algorithms for calculating the shortest paths when they are correctly coded in C programming language. The eight algorithms that we selected for this experiment are the most efficient, either measured in terms of worst–case bounds or marked as such from previous computational studies; they include the redistributive heap algorithm due to Ahuja et al. (1988). We suggest computer implementations that take full advantage of this programming language. In fact, the network representation and the various data structures used to keep the scan eligible list may be managed by using only additions and **no** multiplications, and this is not possible with FORTRAN. These capabilities unique to C yield several interesting conclusions: one may expect to speed up a shortest path algorithm by a factor of 20% and, in some cases, this factor may reach 30%. Furthermore, we infer that similar results may be obtained for other network algorithms as well.

KEY WORDS: SHORTEST PATHS,
REDISTRIBUTIVE HEAP,
C PROGRAMMING LANGUAGE,
POINTER AND ARRAY IMPLEMENTATIONS.

RESUME

Ce papier évalue les performances des algorithmes pour la détermination des chemins les plus courts lorsqu'ils sont programmés efficacement en langage C. Nous avons choisi huit algorithmes parmi ceux qui ont été jugés les plus efficaces d'après des études tant numériques que théoriques. Notre étude inclut l'algorithme récent de Ahuja et al. (1988) qui fait usage d'un *redistributive heap*. Nous suggérons des implantations qui exploitent les possiblités du C. En particulier, le réseau ainsi que les structures de données utilisées pour maintenir la liste des sommets candidats peuvent être manipulés sans multiplication, ceci étant impossible en FORTRAN. Ces possibilités uniques au langage C nous permettent d'observer des améliorations allant de 20% à 30% dans les temps d'exécution des algorithmes du chemin le plus court. Finalement, nous croyons que des résultats similaires peuvent être obtenus pour d'autres algorithmes de réseau.

MOTS–CLES: CHEMINS LES PLUS COURTS,
REDISTRIBUTIVE HEAP,
LANGAGE C,
IMPLANTATIONS AVEC DES TABLEAUX ET DES POINTEURS.

I. Introduction

Shortest path algorithms play a central and capital role in several research and application areas: network analysis, transportation planning, telecommunications, computer science, etc. A large gamut of problems may be formulated as shortest path models. The solution of an even larger set of problems requires that shortest path problems be solved repeatedly as subproblems. Thus, the question: *which is the fastest shortest path algorithm* has a great importance, from both theoretical and practical standpoints. And this question is still very much actual.

New contributions are continuously appearing in the shortest path literature; Pallottino (1979), Glover et al. (1985a, 1985b), Fredman and Tarjan (1987), Ahuja et al. (1988), for example. In particular, the new algorithms, due to Ahuja et al. (1988), offer the best theoretical worst–case bounds. Unfortunately, we were not able to find computational results for these algorithms.

It is noteworthy that recent algorithms are essentially based on implementations of particular data structures and on efficient methods for processing them. This emphasizes the impact of the programming language on the algorithmic efficiency.

There have been many excellent computational studies concerning shortest path algorithms. The list includes Gilsinn and Witzgall (1973), Denardo and Fox (1979), Dial et al. (1979), Glover et al. (1985a, 1895b), Gallo and Pallottino (1988) and Hung and Divoky (1988), to name but a few. We notice that all numerical experiments reported in these papers were conducted with procedures programmed, almost exclusively, in FORTRAN. One exception: the paper of Denardo and Fox (1979) reports results with implementations in assembly programming language (COMPASS) for use on a CDC Cyber computer.

The major limitation of the empirical approach used to evaluate the computational performance of algorithms is that, in general, any extrapolation of results is only valid for the computer(s) and computer language(s) used, and the problems tested. Thus, since the C programming language offers more and richer models of data representation and computation than FORTRAN, any extrapolation of previous results must be considered speculative. The question of how algorithms coded in C perform, both in absolute terms and

when compared among themselves, is still open. We aim at answering some of these questions.

The main purpose of this paper is to analyze the performance of algorithms for calculating the shortest paths from one node to all other nodes in a directed network, when they are correctly coded in C programming language.

The choice of C is mainly justified by the efficiency of the code it produces, and by the control of complex data structures (in contrast to FORTRAN) and by the manipulation of memory addresses (in contrast to Pascal) that it offers. As we will also see in the following sections, it is possible to access data without making use of multiplications; this is not possible with FORTRAN, for example. Another advantage of C resides in its portability since, although C exploits all the capabilities of its host computer, it is independent of any particular machine architecture (Kerningham and Ritchie, 1988).

We suggest computer implementations that take full advantage of the capabilities of this programming language and we compare the performance of our programs with that of traditional implementations. The eight algorithms that we selected for this experiment are the most efficient, either measured in terms of worst–case bounds or marked as such from previous computational studies; they include the redistributive heap algorithm due to Ahuja et al. (1988).

The paper has the following organization. Section II contains the necessary notations and definitions. We also discuss how graphs are efficiently represented and manipulated by using a pointer (in C), or a traditional, array (FORTRAN like) implementation. The various algorithms, their corresponding data structures and manipulation strategies, are presented in Sections III and IV together with their most efficient implementations in the two contexts. The results of experiments conducted on networks based on three different graph topologies are described in Section V. Section VI summarizes the findings.

II. Data Structures and Algorithms

Let $N = (V, A, L)$ denote a *network* over a directed graph $G = (V, A)$ with $n = |V|$ nodes and $m = |A|$ arcs, and a set of functions L, defined on the graph. For the problems that we consider in this paper, the set L consists of a length function $l : A \rightarrow R$, where R is the set of real numbers. If $u = (i,j) \in A$, then the number l_u will be called the *weight* or *length* of arc u. We also use l_{ij} to denote the length of arc (i,j). We shall denote by l_{max} the maximum arc weight in the network. The set of all arcs emanating from node i is called the *forward star* of i, and is denoted by Λ_i. An *(r,s) path* is a sequence of arcs of the form (r,i_1), (i_1,i_2), ..., (i_k,s). The length of an (r,s) path is the sum of the lengths of its arcs. A *directed cycle* is an (r,r) path containing at least one arc in which no node, except node r, is repeated. The arc lengths may be either positive or negative. The only assumption we need to make is that there are no directed cycles with negative lengths in N. In the following, we shall simply use *path* instead of (r,s) path.

A prototype method for the single source shortest path problem, which is based on the exploration of complete forward stars, may be stated as follows (Gallo and Pallottino, 1986):

Prototype Algorithm

1. Initialize $d_r = 0$, $p_r = nil$ and $d_i = \infty \ \forall\, i \neq r$.
   Define the scan eligible list $Q = \{r\}$.

2. If $Q = \emptyset$, STOP. Otherwise **select** (and **remove**) a node i from Q.

3. Scan the forward star of node i, in order to adjust the node labels and to test the optimality conditions.
   Insert into Q any successors of node i for which the label has been improved.

4. Go to step 2.

How nodes are inserted, selected and removed from Q directly affects the way in which the graph G is explored in order to achieve optimality. Generally speaking, algorithms may be divided into two classes: (i) methods that may be applied to networks with nonnegative arc lengths and that always remove the node with the minimum label from Q;

(ii) algorithms that apply to networks with negative arc lengths but without negative cycles, and that only consider the position of a node in Q. The first group of algorithms is known as *Label–Setting*, and corresponds to *shortest–first search* methods (Gallo and Pallottino, 1986) for exploring the graph. On the other hand, the second group is made of *Label–Correcting* algorithms and represents methods where *list–search* strategies (either *breadth-first* or *depth-first*, or any other search strategy derived from them; see Gallo and Pallottino, 1986), are used to explore the graph.

Besides the basic algorithmic approach, the computer representation of the forward star of a graph and the efficiency of the procedures used to scan it are the major elements one has to consider when developing efficient implementations of shortest path algorithms. This conclusion equally applies to most other network algorithms.

Several data structures may be used to represent graphs and networks within the computer memory (Gondran and Minoux, 1985). Yet, in practice, their implementation efficiency may significantly vary. Since the shortest path methods which offer the most interesting performances in terms of their computational behavior are based on the scanning of the forward star, the data structures that insure their most efficient implementations are based on the representation of the adjacency matrix of the graph (Gallo and Pallottino, 1986; Aho et al., 1987; Dial et al., 1979).

In this representation, all arcs that begin at the same node are stored together, and each is represented by a recording of only its end node and length. This approach, called the *forward star form* or the *arc–list structure* or the *adjacency structure* requires, for each node, a pointer to indicate the block of computer memory locations for the arcs beginning at this node. Thus, only $n + 2m$ words of memory are needed to maintain the network.

It is part of the folklore that, when programming in FORTRAN, nodes and arcs are represented by the first n and m integers respectively. Consequently, the most efficient way to implement the adjacency structure is to use a triplet of arrays A[], ND[] and LNGTH[] (Dial et al., 1979; Gallo and Pallottino, 1988). The dimension of array A[] is n; that of arrays ND[] and LNGTH[] is m. The information relative to A_i is stored in ND[] and LNGTH[], from position A[I] to position A[I + 1] − 1. Additionally, two arrays of

dimension n, P[] and D[], are defined to hold the shortest path tree and the shortest distance vector respectively, which are returned by the algorithms. In this *array representation*, the labeling step of node i is achieved by the sequence shown in Figure 1a.

```
A1    START = A[I]
A2    END = A[I+1]-1
A3    DI = D[I]
A4    DO 10 K=START, END
A5       J = ND[K]
A6       DJ = DI + LNGTH[K]
A7       IF (DJ .LT. D[J]) THEN
A8          D[J] = DJ
A9          P[J] = I
A10      ENDIF
A11 10 CONTINUE
```

Figure 1a. Array Implementation

```
C1    start = i→a;
C2    end = (i + 1)→a;
C3    di = i→d;
C4    for (k=start; k<end; k++) {
C5       j = k→nd;
C6       dj = i→d + k→lngth;
C7       if (dj < j→d) {
C8          j→d = dj;
C9          j→p = i;
C10      }
C11   }
```

Figure 1b. Pointer Implementation

Figure 1. Labeling Step

Since the location of A[I] in the computer memory is obtained by an operation of the form (Knuth, 1973; p. 240):

$$LOC (A[I]) = BASE + (I * NBYTE)$$

where BASE is the base address for the array A[] and NBYTE is the size of each element of A[]. The access to the data needed to scan Λ_i and to test the optimality conditions requires $3 + 4 \times |\Lambda_i|$ additions and $3 + 4 \times |\Lambda_i|$ multiplications.

A pointer is a variable that contains the address of another variable. The C programming language offers the capability to manipulate pointers in ways that depend on what they are pointing to. For example, if we have a pointer on a 32-bit integer and we increment the pointer, it then points to the next integer (4 bytes away). Thus, in C, pointers are an excellent way to do array indexing (see Kernighan and Ritchie, 1988). Note that, by this capability, C contrasts with Pascal for which pointers are dynamic objects that can be set only by calling a storage allocation function. For the Pascal programming language, and all other programming languages that do not provide pointer arithmetic (such as FORTRAN, Basic, ...), the adjacency structure is best implemented as described above.

The fact that the arrays A[], P[] and D[] and the arrays ND[] and LNGTH[] have the same dimension respectively, invites us to consider a new data organization when programming in C. The nodes and arcs are represented by contiguous memory locations where their records are stored. The following instructions in C that declare the two structures, called NODE and ARC, designed to hold these data are shown in Figure 2.

```
struct NODE {
      struct ARC *a;       /* array A[] */
      struct NODE *p;      /* array P[] */
      int d;               /* array D[] */
}
struct ARC {
      struct NODE *nd;     /* array ND[] */
      int lngth;           /* array LNGTH[] */
}
```

Figure 2. Forward Star Implementation in C

Then, the labeling step of node i is implemented in C by the sequence of instructions shown in Figure 1b.

It is important to notice that, due to the pointer capabilities of the C programming language, the scanning of the forward star of a node and the testing of the optimality conditions require only $3 + 4 \times |\Lambda_i|$ additions and no multiplications. Therefore, independently of the type of algorithm chosen, we expect implementations of shortest path algorithms in C to be more efficient than the corresponding traditional ones.

In the following two sections, we discuss the implementation of eight algorithms among the most efficient, either measured in terms of worst–case bounds or marked as such from previous computational studies; they include the redistributive heap algorithm due to Ahuja et al. (1988).

III. Label–Correcting Algorithms

The simplest way to keep the nodes in the scan eligible set Q is to use a list. A *list* is an ordered sequence of n items $i_1, i_2, ..., i_n$. The item i_1 is the *head* of the list and i_n is its *tail*.

A *queue* is a list for which additions are allowed only at the tail and deletions only at the head. Queues are used to implement the well known shortest path method credited to Ford (1956), Moore (1959) and Bellman (1958). This method, which we refer to as LQUEUE, processes nodes in a "first–in first–out" manner, that is, the graph is explored level by level. In fact, all shortest paths of one arc are first computed, followed by those made up of two arcs, and so on. Since the number of arcs in a shortest path is bounded by n, the computational complexity of LQUEUE is $O(mn)$.

A *stack* is a list for which all insertions and deletions are made at the head of the list. Shier and Witzgall (1981) showed that a shortest path algorithm for which Q is implemented as a stack is not polynomially bounded. A *deque* (double-ended queue, Knuth, 1973) is a list for which additions and deletions are possible at either end. A deque may be viewed as a stack and a queue connected in series in such a way that the tail of the stack points to the head of the queue. Deques are used to implement the two-way criterion idea attributed to D'Esopo and Pape (see Pape, 1974, 1980). This method, called LDEQUE, adds a node at the head of Q if it has already appeared on the list before; the node is appended at its tail otherwise. Nodes are always removed from the head of the scan eligible node list. When all nodes have a finite label, the method behaves like an algorithm that uses a stack, the computational complexity of LDEQUE is then $O(n2^n)$.

Pallottino (1979) suggested the use of a *2queue*, instead of a deque, in order to implement the two-way criterion method. A *2queue* can be interpreted as two queues connected in series. Unlike LDEQUE, this implementation of the two–way criterion is polynomially bounded ($O(mn^2)$). We refer to this method as L2QUE.

Recently, Glover et al. (1985b) introduced a new criterion to control the insertions of nodes into the scan eligible node list. This method, called *Partitioning Shortest Path Algorithm*, divides the list Q into two distinct lists, NOW and NEXT. A threshold value t is

computed to determine what nodes currently in NEXT may be moved to NOW. At each iteration, a node i is chosen in the NOW list in a "last–in first–out" way. After the NOW list is exhausted, a new value for t is computed and qualifying nodes in NEXT are transferred to NOW. The process is repeated until NEXT is empty. Glover et al. (1984) showed that any partitioning shortest path procedure is polynomially bounded. Based on this approach, different algorithms may be obtained according to the formula used to compute the threshold value. The method we had implemented, called LTHRS, is based on the procedure THRESHX2 (Glover et al., 1985b) and on the implementation strategy developed by Gallo and Pallottino (1988). This implementation is $O(mn^2)$, while the original implementation of Glover et al. is $O(mn)$. The difference is due to the manner in which the NOW list is treated. Glover et al. (1985b) use, in the labeling step, a third list, NOW', to receive the new qualified nodes; on the other hand, Gallo and Pallottino (1988) insert these nodes directly at the tail of NOW leaving a copy in NEXT (when NEXT is scanned, copies are deleted). In practice, however, this does not seem to affect the performance of the algorithm.

In order to implement each of these four methods properly and efficiently, linked lists have to be used.

A *linked list* is an ordered sequence of items, where each item contains a link to the next item of the list. When the nodes of the graph are represented by the first n integers, the linked list is efficiently implemented by means of an array Q[] of dimension $n+1$ (see Figure 3).

$$Q[I] = \begin{cases} 0 & \text{if } i \text{ is not in } Q \\ J & \text{if } i \text{ precedes } j \text{ in } Q \\ N+1 & \text{if } i \text{ is the last element of } Q \end{cases}$$

and $Q[N+1] =$ the head of Q.

Figure 3. Linked List Implementation by Means of an Array Q[]

Since, in the C programming language, nodes may be represented by memory addresses, the structure NODE, shown in Figure 4, is used to implement shortest path algorithms that use linked lists.

```
struct NODE {
    struct ARC *a;
    struct NODE *p;
    int d;
    struct NODE *q;
};
```

Figure 4. NODE Structure for Linked List Implementation

Procedures that perform additions and deletions by using this structure and these variables are shown in Figure 5. Note that the pointer implementation of the same instructions is displayed as a commentary, to the right of the figure. We emphasize again that this list implementation requires only additions and no multiplications, in contrast to the FORTRAN–like one. Yet, the difficulty level of implementing the two structures is equivalent.

addition at the head

```
j → q = n1 → q;                 /* Q[J] = Q[N+1] */
n1 → q = j;                     /* Q[N+1] = J    */
if (last == n) last = j;
if (pntr == n) pntr = j;        /* only for L2QUE procedure */
```

addition at the tail

```
last → q = j;                   /* Q[LAST] = J   */
j → q = n1;                     /* Q[J] = N+1    */
last = j;
```

deletion (always at the head)

```
i = n1 → q;                     /* I = Q[N+1]    */
n1 → q = i → q;                 /* Q[N+1] = Q[I] */
i → q = nil;                    /* Q[I] = 0      */
if (last == i) last = n1;
if (pntr == i) pntr = n1;       /* only for L2QUE procedure */
```

Figure 5. Linked List Manipulations with Pointers

IV. Label–Setting Algorithms

A *priority queue* is an abstract data structure that is used to keep a collection of items, each of which has associated a real value label. Several operations are possible on a priority queue. These operations are used by label–setting algorithms:

INSERT (i,Q) Insert item i into priority queue Q, not previously containing i.

REINSERT (i,Q) Reinsert item i into priority queue Q after the label of i has been modified.

DELETEMIN (Q) Delete and return an item of minimum (or maximum) value label from priority queue Q.

The idea of using a priority queue in order to compute shortest paths is credited to Dijkstra (1959). In terms of the operations described above, the algorithm may be stated as follows:

Dijkstra algorithm

1. Initialize $d_r = 0$, $p_r = nil$ and $d_i = \infty \; \forall \, i \neq r$.
   Define the priority queue $Q = \{r\}$.

2. If $Q = \emptyset$, STOP. Otherwise $i \leftarrow$ DELETEMIN(Q).

3. Scan the forward star of node i in order to test the optimality conditions.
   If the label of node j is improved for the first time then INSERT(j,Q),
   else REINSERT(j,Q).

4. Go to step 2.

It can be easily shown that, if all arc costs are nonnegative, then Dijkstra's algorithm performs exactly n DELETEMIN operations.

The simplest way to manage a priority queue is to use an ordered linked list that contains all the nodes in the priority queue. In this case, the DELETEMIN operation consists in scanning the list in order to find the minimum label node. A variant uses an unordered linked list. In the literature, these methods have been proven to perform poorly on almost

every graph topology (see, for example, Gallo and Pallottino, 1988 or Mondou, 1989). For this reason, this paper does not cover the linked list implementations of priority queues.

In fact, both ordered and unordered linked lists display the property that the number of operations needed to insert or delete nodes is a linear function on the number of nodes in the priority queue. This implies a $O(n^2)$ theoretical complexity. A significantly more efficient priority queue implementation is to use either heaps or buckets.

4.2.1 d–heaps

A *heap–ordered tree* is a tree where each node corresponds to one item of a priority queue and the value label of each node is greater than (less than) or equal to the value label of its parent. By the transitivity law, the root contains a minimum (maximum) value label item. A *d–heap* is a heap–ordered tree for which each node has at most d children and nodes are added in breadth–first order. The time for inserting an item in a d-heap of k items is $\log_d k$; for deleting an item, it is $d\log_d k$. Thus, the total time needed by an implementation of the Dijkstra algorithm that makes use of a d–heap is $nd\log_d n + m\log_d n$. The value of d that minimizes the total bound for the algorithm is a solution of the equation $d(\log_e d - 1) = m/n$, and good approximations are either $d = \max\{2, m/n\}$ or $d = 2+m/n$ (Mondou, 1989; Tarjan, 1983). On the other hand, Johnson (1972) provides empirical evidence that by setting $d = 2$ provides the fastest heap structure for sparse problems. This may be explained by the fact that the computational bound indicated above does not take into account the constants that are involved in heap operations. Also, it is worthwhile to round the value of d to the nearest power of 2 so that multiplications and divisions may be achieved via binary shifts. For these reasons, the code used in this study makes use of a binary heap.

A binary heap is efficiently implemented on a digital computer by using a pair of arrays HP[] and Q[] of dimension n. The cell HP[K] contains the node number placed in position K of the heap. The array Q[] plays the role of the heap directory:

$$Q[I] = \begin{cases} K & \text{if HP[K] = I} \\ 0 & \text{if } i \text{ is not in } Q. \end{cases}$$

Thus the children of heap node K are the cell numbers 2K and 2K+1. The parent of a

node K is simply ⌊K/2⌋. Finally, the variable NHP is used to keep track of the heap size (the number of nodes currently in the heap).

Pointer multiplication and pointer division are not allowed in the C programming language. For this reason, there is no real need for the elements in Q[] to be pointers. However, an efficient implementation is done by using the following variables shown in Figure 6. The procedures used to manipulate binary heap structure in C are displayed in Figure 7.

```
struct NODE {
    struct ARC *a;
    struct NODE *p;
    int d;
    int q;
};
struct NODE *hp[];
```

Figure 6. NODE Structure for Binary Heap Implementation

The main advantage of this approach over traditional implementations is due to the fact that heap manipulations often need to access the value D[HP[K]]. In C, this information is obtained more rapidly via pointers.

It is worth mentioning that Fredman and Tarjan (1987) have recently proposed a new heap structure for implementing priority queues. This structure, called the *Fibonacci heap*, produces shortest path algorithms with very good computational complexity. However, it requires a very sophisticated implementation and considerable memory space (7 arrays of dimension n). Besides, as reported by Hung and Divoky (1988), it is disappointingly slow. Consequently, we have not considered it in our study.

4.2.2 Buckets

The second popular data structure used to keep items in priority queues is a *bucket* structure. For a long time, the algorithm developed by Dial (1969) was considered as the fastest priority queue implementation for the Dijkstra algorithm. Since then, different

removal of the minimum label node from Q

```
i=hp[1]; i->q=0; min=i->d;
x=hp[nhp]; dummy=x->d; nhp--;
if (nhp) {
   h1=1;
   do {
     h2=(h1<<1);
     if (h2<=nhp) {
       if( (h2<nhp) && (hp[h2+1]->d<hp[h2]->d)) h2++;
       j=hp[h2];
       if (j->d<dummy) {
          hp[h1]=j;
          j->q=h1;
          h1=h2;
       }
     }
   } while(h1==h2);
hp[h1]=x; x->q=h1;
```

insert – reinsert

```
if (!j->q) {
  nhp++; hp[nhp]=j; j->q=nhp;
}
h1=j->q; h2=(h1>>1);
while ( (h2) && (j->d<(x=hp[h2],x->d)) ) {
   hp[h1]=x;
   x->q=h1;
   h1=h2; h2=(h1>>1);
}
hp[h1]=j; j->q=h1;
```

Figure 7. Binary Heap Manipulations in C

studies have shown that Dial's method is dominated by other algorithms (see Dial et al. 1979; Glover et al. 1985a, 1985b; Hung and Divoky 1988). However, all studies were based on FORTRAN implementations of the algorithm. We may think that it is worthwhile to implement Dial's method in C since, the poor performance of SDIAL is attributed to the scanning of a long array Q[], in order to find the next first non–empty bucket. This operation in C is efficiently done. This code will be referred to SDIAL.

Denardo and Fox (1979) devised a k–level hierarchy of buckets, for a fixed k. The method displays a good theoretical efficiency. However, we have not included it in our study because it has a high implementation level of difficulty and it shows a poor empirical performance (see Divoky and Hung, 1988). Yet, we implemented a fixed bucket method,

which we refer to as SBCKT, especially designed to be efficient for a large value of l_{max}. The number of buckets is fixed to 128 and the bucket size to 64. Since we used values that are of the power of 2, binary shifts are used for dividing and bit mask for counting.

Recently, Ahuja et al. (1988) suggested a very simple and elegant use of buckets called *Redistributive Heaps*. The shortest path algorithms using redistributive heaps have the best known computational complexity. In order to verify their computational efficiency, we implemented the redistributive heap at one level.

Independently of the strategy that is used to insert nodes into buckets, we need an efficient way to store nodes into buckets. Denardo and Fox (1979) analyzed three schemes for coding buckets on a digital computer. The first one employs double linked lists for linking nodes in buckets, and a master list containing pointers to the head of each bucket. They found that this method dominates the two others and this is now a part of the computer science folklore. In FORTRAN this scheme is implemented by means of a pair of arrays UP[] and DOWN[] of dimension $n+1$, and an array Q[] of dimension equal to the number of buckets used. The special head–of–bucket symbol is implemented by using the sign bit of UP[J] and the tail–of–bucket symbol is N+1. The utilization of these special symbols for the head and the tail of buckets eliminates the need to test for special cases. The data structure is shown in Figure 8.

$$Q[PNTR] = \begin{cases} J & \text{if } j \text{ is the first element of bucket PNTR} \\ N+1 & \text{if bucket PNTR is empty} \end{cases}$$

$$UP[J] = \begin{cases} 0 & \text{if } j \text{ is not in the priority queue} \\ J & \text{if } i \text{ precedes } j \text{ in the same list} \\ -PNTR & \text{if } j \text{ is the first element of bucket PNTR} \end{cases}$$

$$DOWN[J] = \begin{cases} 0 & \text{if } j \text{ is not in the priority queue} \\ J & \text{if } i \text{ follows } j \text{ in the same list} \\ N+1 & \text{if } j \text{ is the last element of a bucket} \end{cases}$$

Figure 8. Buckets – Array Implementation

To implement this structure in C by using pointers a number of minor adjustments have to be made since C does not permit to mask pointers. So, instead of using the sign bit of UP[J], we use the capability offered by C to store different types of data in the same memory location. This is done by using the C *union* instruction and a variable *flag* that indicates if node *i* is or is not the first element of a bucket. Figure 9 displays this structure.

```
struct node {
    struct arc *a;
    struct node *p,*down;
    union {
        struct node *up;
        struct bucket *pntr;
    } uval;
    int flag,
    int d;
};
struct bucket {
    struct node *head;
};
```

Figure 9. Buckets – Pointer Implementation

In that case, the deletion and insertion of nodes into buckets are achieved by the code sequences shown in Figure 10.

Mondou (1989) also analyzed the use for coding buckets, of single linked lists and a directory of pointers as described in Denardo and Fox (1979). However, it requires a higher number of memory accesses; algorithms programmed following this scheme of implementing the bucket structure are slower than the one described above.

deletion of node j from bucket pntr

```
flag=j->flag;
jdown=j->down;
if (flag) {
  jup=j->uval.up;
  jup->down=jdown;
  j->flag=1;
}
else {
  pntr=j->uval.pntr;
  pntr->head=jdown;
  j->flag=0;
}
jdown->flag=0;
```

insertion of node j into bucket pntr

```
jdown=j->down=pntr->head;
pntr->head=jdown->uval.up=j;
j->uval.pntr=pntr;
jdown->flag=1;
j->flag=0;
```

deletion of the first node of bucket pntr

```
i=pntr->head;
idown=pntr->head=i->down;
idown->uval.pntr=pntr;
i->uval.up= i->down=nil;
i->flag=1;
idown->flag=0;
return(i);
```

Figure 10. C Bucket Manipulation Procedures

V. Computational Results

In this section, we describe the numerical experiments that we conducted in order to answer the three questions that prompted our research effort:

1. Is an implementation in C significantly more efficient than a traditional, FORTRAN–like one?
2. Does the fact that the algorithms are coded in C modify their known relative computational efficiency order?
3. How do algorithms based on redistributive heaps compare with others?

To achieve the first objective, it is not sufficient, however, to simply compare the performances of FORTRAN and C codes of the same algorithms. First, the two programming languages do not have the same set of instructions to direct the flow of execution. Standard FORTRAN, for example, does not provide the *while* instruction, while making heavy use of the redirectional instruction *goto*. Also, it is well known that C compilers produce faster codes than FORTRAN compilers. On the other hand, our objective is not to stage a C – FORTRAN competition, but rather to compare a true pointer implementation of shortest path algorithms, possible in C, to a more traditional, array–based one, which is the only one possible with the most widely accepted high level programming languages (FORTRAN, Pascal, etc.).

We therefore used the ability of C to mimic FORTRAN, and we programmed the array implementation of the algorithms in C. Special care has been taken to insure that an equally efficient coding is achieved for both implementations. In particular, we closely followed the *Pidgin Pascal* descriptions of algorithms of Gallo and Pallottino (1988). Consequently, computing efficiency now exclusively depends on the type of data structure that each class of programming language offers, and comparisons are meaningful.

Tests have been conducted on networks based on graphs belonging to three distinct topological groups:

[1] One set consists of $p \times q$ rectangular *grid networks*. A grid network has pq nodes and $4pq - 2p - 2q$ arcs, and each node is adjacent (connected by arcs) to only the four nodes (if present) to its right, left, above, and below.

[2] The second set of problems are *random networks*. These are networks which are built by randomly selecting two nodes that are then connected to form a new arc. Nodes are selected by using a uniform probability distribution, subject to the following restrictions: the two nodes have to be different, and arcs are not allowed to be duplicated. Special care has been taken to insure that our problem generator always produces strictly connected networks.

[3] The last topologically distinct group of problems consists of *complete networks*. A complete network of n nodes is a network in which each node is adjacent to all others. Thus, the number of arcs in a complete network is $n(n-1)$.

These problems were generated by using a uniform probability distribution with two distance ranges for the arc lengths: [1-100] and [1-10000]. The network generator we developed is based on results by Latour (1987).[1]

The experiments were conducted on a SUN 3/180. The CPU time is measured in 1/60 seconds. In the tables, we report the total CPU time needed to solve ten shortest path problems for which the origins were randomly selected. As usually, I/O and function calls are excluded from the computation of the total time.

All tables in this section share the same format; namely, for each algorithm, the entries on the first line correspond to computing times obtained with the array implementation, while the second line indicates the corresponding times achieved by using the pointer implementation.

5.1 Complete Networks

Since complete networks require an enormous amount of memory space, we restricted this study to small networks with 25, 75, 125 and 175 nodes, respectively. Results are summarized in Table 1.

---

[1] In order to insure its portability, the network generator employs the random number generator described in Bratley, Fox and Schrage (1983). It is available upon request from the authors of this paper.

Algorithms	$l_{max} = 100$				$l_{max} = 10000$			
	n=25	n=75	n=125	n=175	n=25	n=75	n=125	n=175
L2QUE	10	79	274	497	12	100	378	748
	7	68	216	391	8	73	292	570
LDEQUE	9	86	297	517	11	105	389	779
	7	63	223	408	8	82	313	590
LQUEUE	7	64	190	378	11	78	251	480
	7	49	149	293	7	63	192	396
LTHRS	6	40	108	204	6	43	134	270
	5	32	89	160	3	29	86	215
SBCKT	7	57	153	279	7	49	124	267
	7	47	126	236	6	41	104	215
SDIAL	8	47	123	232	13	50	126	236
	6	36	91	171	13	39	96	177
SHEAP	8	50	132	245	8	53	134	250
	6	34	90	168	6	36	90	173
SRHEAP	9	37	123	228	10	56	126	245
	7	39	89	166	12	43	105	186

Table 1. Computational Results – Complete Networks

Note that, for complete networks, Glover et al. (1985a, 1985b) have not suggested any specific value for their parameter X2. However, Gallo and Pallottino (1988) tested different values and they concluded that by using X2 = 0.25 the algorithm LTHRS achieves its best performance. We therefore used this value when testing.

Regarding the performances of the two implementations of the algorithms, on complete networks, the following observations may be made:

- All list–search algorithms (except LTHRS, which behaves as a label–setting method) are dominated by shortest–first algorithms. This is credited to the fact that very dense graphs have a high number of alternative paths between any two nodes. Furthermore, since the arc lengths are integer values, the number of different lengths of these alternative paths increases with l_{max}. This is reflected in the CPU times shown in Table 1; compare the last four columns to the first four, for the first three algorithms.

- SRHEAP is generally not the best algorithm. However, as the size of the network increases, its efficiency becomes comparable to the fastest ones. We infer that for very

large complete networks this algorithm will display the best performance.

- In general, LTHRS is the fastest code.
- In all cases, the pointer implementation of an algorithm outperforms the array implementation of the same algorithm. The significance of the gains seems to increase with the size of the network. Figures 11, 12 and 13 illustrate these results for the larger size of the problems that we tested. Similar results have been observed for all other problems. Additional details and discussion may be found in Mondou (1989).

5.2 Random Networks

Four classes of random network test problems were considered:

Class A, $n = 1000$, $m = 10000$;
Class B, $n = 1000$, $m = 30000$;
Class C, $n = 3000$, $m = 10000$;
Class D, $n = 3000$, $m = 30000$.

Table 2 shows the results we obtained on these random networks. We observe that (see Mondou, 1989, for additional details and discussion):

- LTHRS is the fastest code for all problems.
- With respect to the other methods, shortest–first algorithms generally dominate list search methods.
- The range of the arc lengths significantly affects the performance of the SBCKT algorithm: for limited length ranges the algorithm performs poorly since, due to the high number of nodes in each bucket, it behaves very much as a classical implementation of Dijkstra's method (see Gallo and Pallottino, 1988; Mondou, 1989).
- The fastest label–setting algorithm is SDIAL and, remarkably, it is less affected by the value of l_{max} for denser random network problems than reported in the literature. This may be explained by the fact that we used a C implementation to mimic FORTRAN. We further discuss this phenomenon in the conclusion.

Algorithms	$l_{max} = 100$				$l_{max} = 10000$			
	Class A	Class B	Class C	Class D	Class A	Class B	Class C	Class D
L2QUE	192	625	222	679	197	740	224	699
	138	496	167	537	152	567	187	556
LDEQUE	189	618	223	642	195	714	234	683
	130	473	146	482	147	572	168	555
LQUEUE	180	520	207	592	190	598	244	679
	135	397	163	451	142	429	159	466
LTHRS	115	242	197	333	114	258	211	374
	88	178	144	244	84	191	141	258
SBCKT	1794	2263	7356	14343	223	534	500	1188
	1410	1799	5776	11462	183	427	394	940
SDIAL	134	292	210	396	173	310	351	447
	107	212	173	316	151	242	329	368
SHEAP	238	396	559	746	237	393	571	744
	187	296	467	618	188	301	472	624
SRHEAP	183	316	425	581	304	426	831	914
	156	247	355	438	248	345	662	732

Table 2. Computational Results – Random Networks

- SHEAP is faster than SRHEAP when the arc lengths may take a wide range of values; SRHEAP is faster in the other case. Note, however, that SRHEAP is never the fastest algorithm of them all.

- In all cases, the pointer implementation of an algorithm outperforms the array implementation of the same algorithm. Figures 14, 15 and 16 illustrate these results for the denser 1000–node networks.

5.3 Grid Networks

We generated grid networks with 2500 nodes, with rectangularities of 50 × 50, 25 × 100, 10 × 250, and 5 × 500. For grid networks the value for X2 is set at 1.5,

as computed by Glover et al. (1985a,1985b). The results are shown in Table 3.

Algorithms	$l_{max} = 100$				$l_{max} = 10000$			
	p = 50 q = 50	p = 25 q =100	p = 10 q =250	p = 5 q =500	p = 50 q = 50	p = 25 q =100	p = 10 q =250	p = 5 q =500
L2QUE	176 128	165 125	169 128	157 120	172 129	184 137	171 121	162 122
LDEQUE	158 114	165 114	164 116	159 108	169 129	169 120	167 114	159 116
LQUEUE	318 241	592 450	1236 958	2216 1535	357 256	615 417	1425 981	2211 1584
LTHRS	178 124	166 124	166 121	164 121	169 120	169 124	166 119	155 116
SBUCKET	913 717	590 502	359 271	287 229	234 192	233 176	225 182	208 184
SDIAL	187 163	198 165	198 169	224 191	823 771	1070 1001	2471 2372	4144 3980
SHEAP	385 314	364 299	301 254	287 231	378 318	365 298	318 243	286 232
SRHEAP	440 327	447 364	484 377	495 399	727 583	727 582	732 595	726 591

Table 3. Computational Results – Grid Networks

Based on these results, the main conclusions concerning the relative efficiency of shortest path algorithms, according to their implementation, on grid networks are:

- Three algorithms, L2QUE, LDEQUE and LTHRS, consistently display the best performance on this network topology class. This confirms the results of Gallo and Pallottino (1988). Note that, since the "two-way" criterion (see Section 4.1) has been specifically designed for grid networks, it is not surprising to find L2QUE and LDEQUE among the most efficient methods.

- Once again, one finds that LTHRS is one of the most efficient procedures. It is worth noting that, at least for this network topology, this method behaves rather similarly to

SBCKT and SDIAL.

- The range of the arc lengths significantly affects the performance of the SBCKT algorithm: for limited length ranges the algorithm performs poorly since, due to the high number of nodes in each bucket, the method behaves very much as a classical implementation of Dijkstra's method (see Gallo and Pallottino, 1988; Mondou, 1989).

- SDIAL is also significantly affected by the range of the arc lengths. However, its behavior is exactly the opposite to that of SBCKT.

- Consequently, according to the arc length range, either SDIAL or SBCKT is the fastest label–setting algorithm. This may suggest the building principle for an improved label–setting procedure using buckets. We further discuss this phenomenon in the conclusion.

- We have also noted that the effect of l_{max} on the efficiency of the SDIAL algorithm is only about half (Mondou, 1989) of what one would expect.

- SHEAP is always faster than SRHEAP, and the gap increases when the maximum arc length increases.

- In all cases, the pointer implementation of an algorithm outperforms the array implementation of the same algorithm. Figures 17, 18 and 19 illustrate these results for the square, 50×50, grid networks.

VI. Conclusions

This study is the first to use the C programming language to evaluate the efficiency of shortest path algorithms, and it yields several rather interesting conclusions. It is also one of the first to perform an empirical study of the practical computational efficiency of the class of shortest path algorithms based on redistributive heaps.

To the first question that prompted our research effort, we provide empirical evidence by using pointers that true C implementations of shortest path algorithms are significantly more efficient than traditional ones (i.e., using arrays).

The improvements in computing times are essentially due to the fact that in C implementations, the forward star of each node may be scanned without performing any multiplication. The C capabilities that explain this property have a second important effect: the data structure used to keep the scan eligible list Q may also be managed (search, insertions, deletions, etc.) by using only additions and no multiplications. Consequently, by using C implementations with pointers, one may expect to speed up a shortest path algorithm by a factor of 20%; in some cases, the speedup factor may reach 30%.

Thus, not all the algorithms are similarly affected; furthermore, for the same type of algorithm, the effect of an implementation in C may be more or less important, according to the network topology. From a general standpoint:

[1] The relative order of the shortest path algorithms, with respect to their actual computational efficiency, is not significantly modified when the algorithms are coded in C. The relative differences among algorithms may differ, but this study has shown that, in this area at least, the conclusions obtained by using C and pointer implementations do not differ from those of previous studies based on FORTRAN and array implementations.

[2] In spite of the fact that it displays the best theoretical efficiency, the redistributive heap algorithm does not emerge as the best method in terms of actual computation efficiency. Algorithms that use thresholds are still, in general, the fastest.

In the following, we summarize the conclusions of our study for each algorithm.

SRHEAP employs buckets but, unlike SDIAL, the number of buckets used is very small, 16, for example, for networks where $l_{max} = 10000$. For dense problems, its efficiency becames comparable to that of the fastest procedures as the size of the network increases. Thus, we may infer that this algorithm will display the best performance for complete, rather large networks. However, even if it has the best theoretical efficiency, in practice, SRHEAP behaves poorly for sparse networks with large values of l_{max}. Briefly, this behavior may be explained by the high number of node redistributions, from bucket to bucket, required by the algorithm. This also suggests that the implementation of algorithms that use redistributive heaps at two levels may not perform any better, since they also require a considerable number of node redistributions from one sub–bucket to another. The interested reader may consult Mondou (1989) for a thorough discussion of this phenomenon.

It is now widely accepted that the threshold algorithms are dominant among shortest path procedures. In this study, we have also found that the LTHRS procedure is the fastest code over all network configurations. For this algorithm, we estimate that pointers accelerate it by a factor of at least 20%.

Algorithms LDEQUE and L2QUE display the same performance. For grid networks, they are to be preferred to LTHRS, since they are equally fast, but are easier to implement. The versions coded in C by using pointers are about 20% faster than traditional implementations. For real problems, however, L2QUE should be used, rather than LDEQUE, since it is polynomially bounded. Finally, in dense problems, LQUEUE is faster than both LDEQUE and L2QUE; however, it is consistently dominated by a label–setting algorithm.

Algorithm SHEAP is the one that takes the greatest advantage of the pointer implementation. This is explained by the relatively high number of array accesses needed to keep the heap property in traditional implementations. On grid networks, SHEAP is the fastest label–setting procedure, and, for this case, the speedup factor is about 30%.

The efficiency of SDIAL increases with the density of the network. This is explained by the fact that in this algorithm π^*_{max} buckets are scanned, where $\pi^*_{max} = \max_i \{ \pi^*_i \}$. Since shortest paths have fewer arcs in dense networks than in sparse networks, the length

of the longest shortest path, π^*_{max}, is small when the network is dense. Thus, for these networks, the value of l_{max} has little effect on the algorithmic efficiency. In fact, we observed that, for sparse problems, the effect of l_{max} on SDIAL is less than reported in the literature: based on the study of Gallo and Pallottino (1988), we calculated that the observed effect of l_{max} is only half the one observed when using FORTRAN codes (Mondou, 1989). Another surprising result is that the use of pointers ameliorates the efficiency of SDIAL by a factor of about 15%. We may explain this behavior by the fact that the scanning of arrays in C is more efficiently performed than in FORTRAN.

Thus, even a traditional implementation of a shortest path algorithm will run faster when coded in C. In this case, a good alternative is to use SBCKT rather than SDIAL. Furthermore, since arrays are efficiently scanned by using pointers and since in C one may perform pointer arithmetic by using shift operations, an efficient implementation of an algorithm based on buckets should use a number of buckets equal to 512 (a power of 2) and a variable bucket size equal to the nearest power of two of the quotient l_{max} / 512.

To conclude, we believe that the implementations described in this paper, and the gains in algorithm efficiency that they generate, may not be obtained with any other commonly used high level programming language, even if they offer dynamic data structures and pointers (such as Pascal). This is possible since C provides unique models to efficiently manipulate computer memory addresses. We have also shown that the level of difficulty required to implement shortest path algorithms in C by using pointers is not greater than that required by traditional implementations that use arrays. Thus, the significant efficiency gains reported in this paper are due to the choice of the programming language and to the proper use of its data structure manipulation capabilities. We expect that similar results may be obtained for other network algorithms as well.

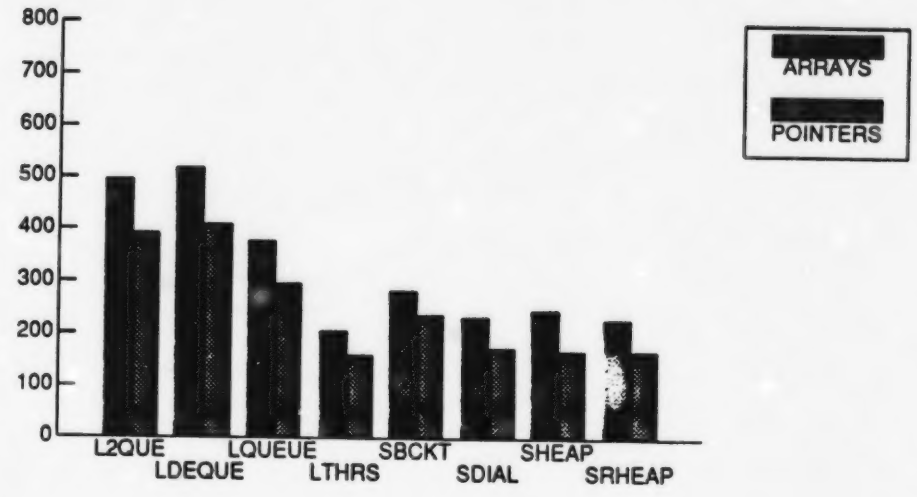

Figure 11. Computation Time of Algorithms on Complete Graphs
(CPU time: 60th of second, n=175, l_{max}=100)

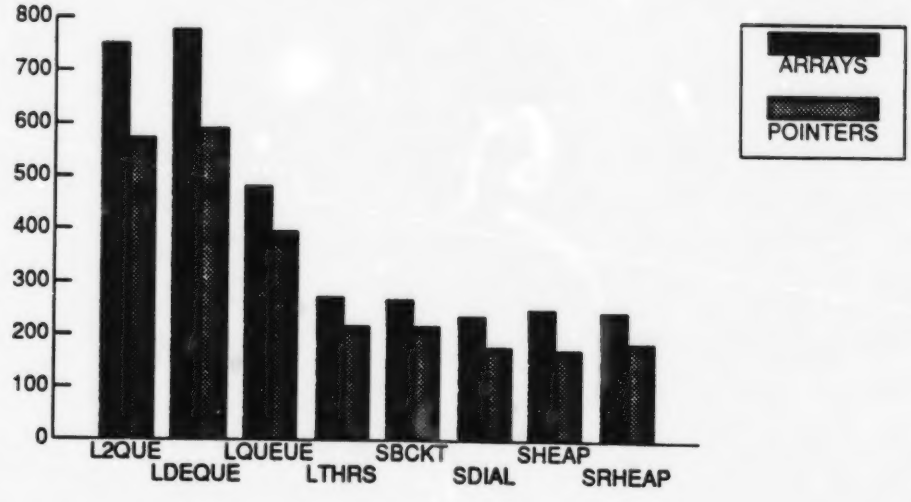

Figure 12. Computation Time of Algorithms on Complete Graphs
(CPU time: 60th of second, n=175, l_{max}=10000)

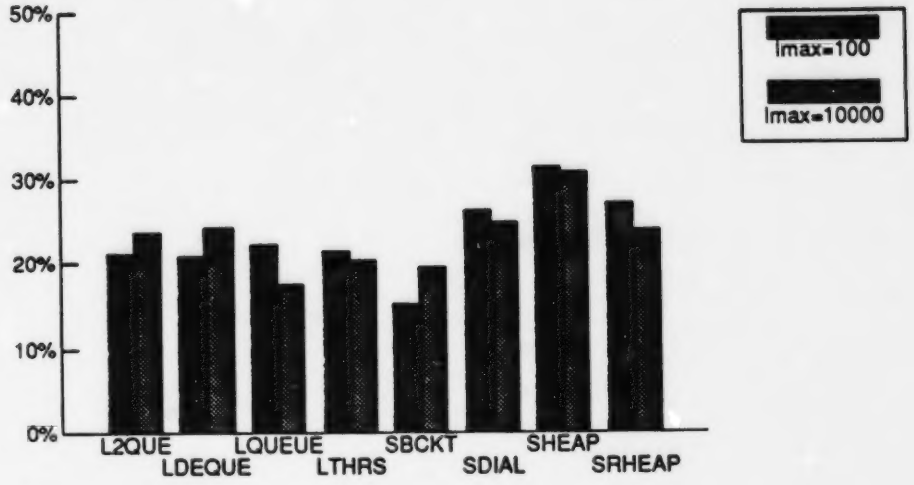

Figure 13. Efficiency Improvements – Pointer Implementation
(Complete Graphs, n=175)

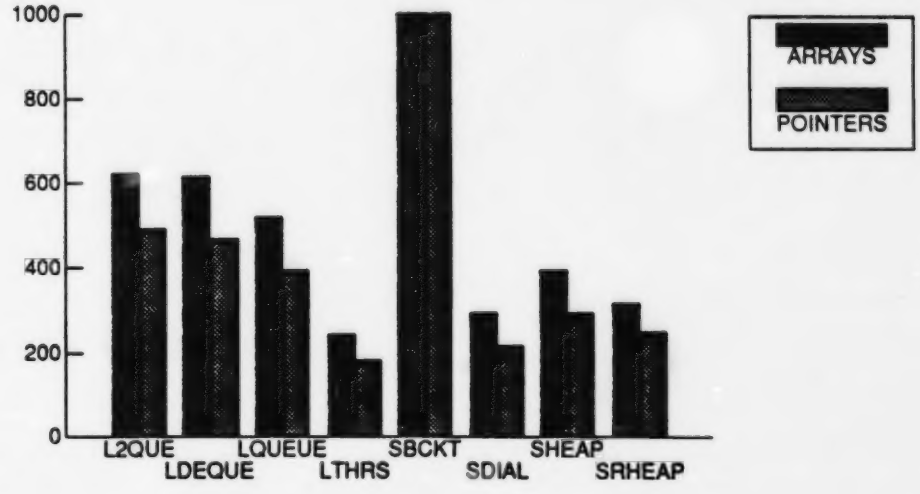

Figure 14. Computation Time of Algorithms on Random Graphs
(CPU time: 60th of second, n=1000, m=30000, l_{max}=100)

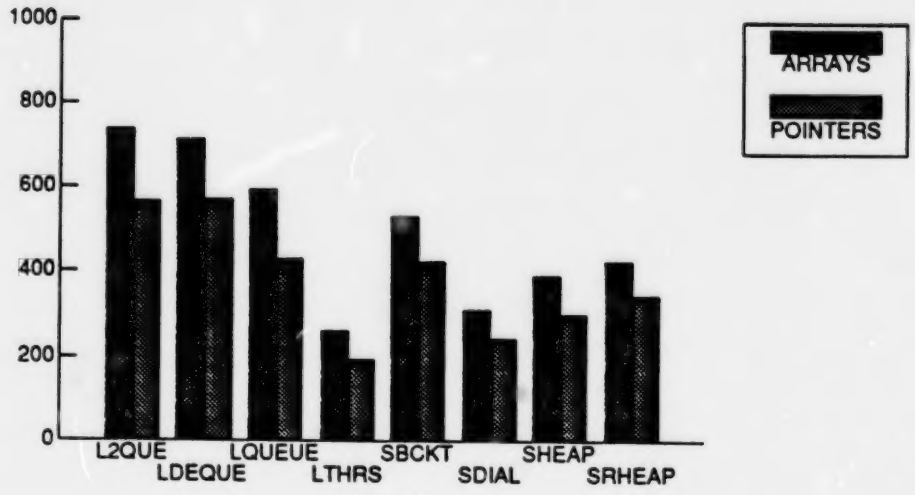

Figure 15. Computation Time of Algorithms on Random Graphs
(CPU time: 60th of second, n=1000, m=30000, l_{max}=10000)

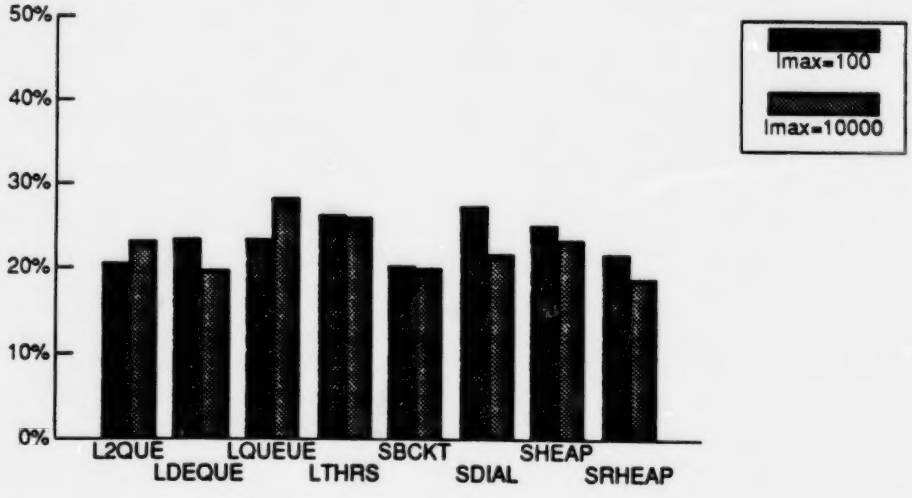

Figure 16. Efficiency Improvements – Pointer Implementation
(Random Graphs, n=1000, m=30000)

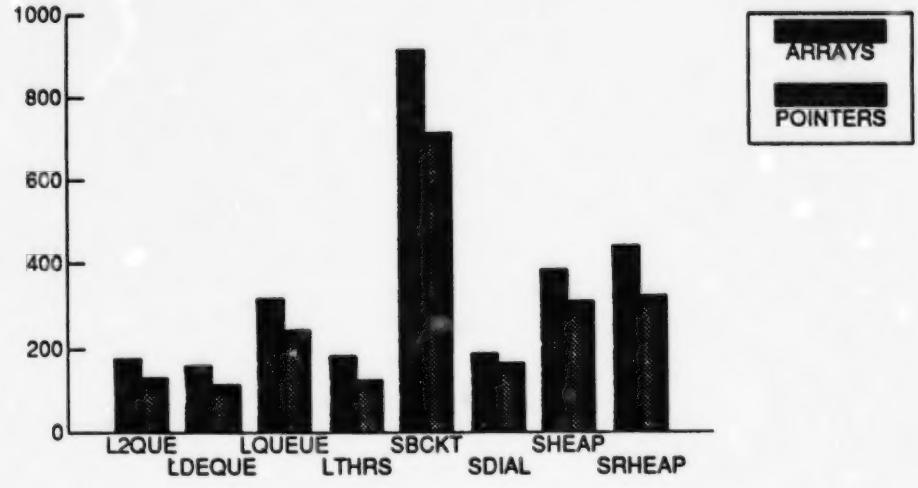

Figure 17. Computation Time of Algorithms on Grid Graphs
(CPU time: 60th of second, p=50, q=50, l_{max}=100)

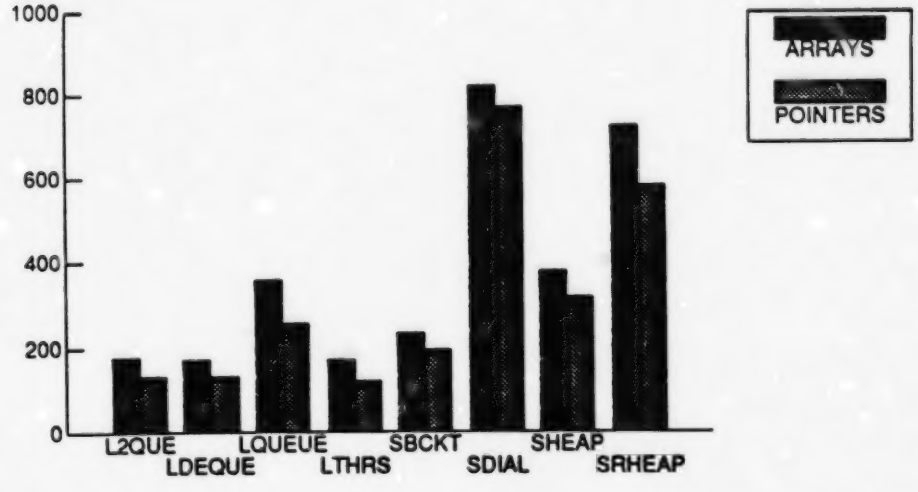

Figure 18. Computation Time of Algorithms on Grid Graphs
(CPU time: 60th of second, p=50, q=50, l_{max}=10000)

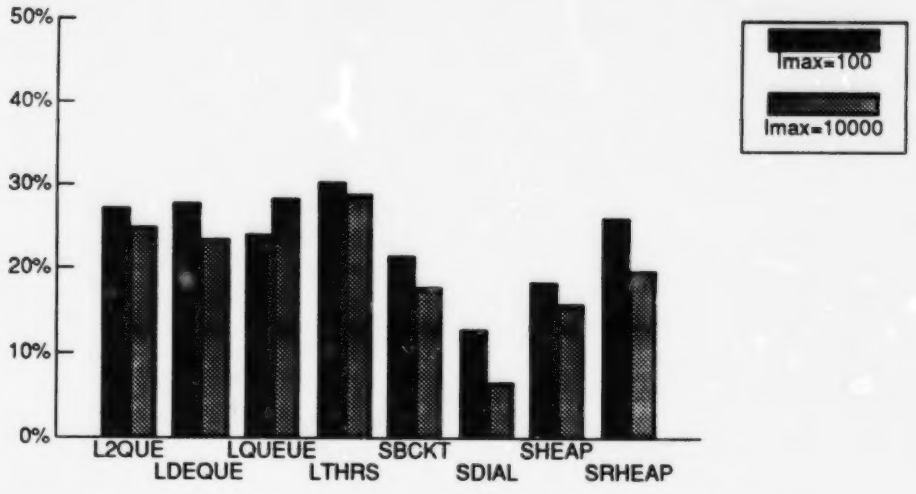

Figure 19. Efficiency Improvements – Pointer Implementation
(Grid Graphs, p=50, q=50)

REFERENCES

Aho, A., Hopcroft, J. and Ullman, J. (1987), **Structures de données et algorithmes**, Addison–Wesley Europe, Inter Editions, Paris.

Ahuja, R.K., Mehlhorn, K., Orlin, J.B. and Tarjan, R.E. (1988), *Faster Algorithms for the Shortest Path Problem*, Technical report No. 193, Operations Research Center, M.I.T.

Bellman, R. (1958), *On a Routing Problem*, **Quarterly of Applied Mathematics**, 16, 88–90.

Bratley, P., Fox, B.L. and Schrage L.E. (1983) **A Guide to Simulation**, Springer–Verlag, New–York.

Denardo, E.V. and Fox, B.L. (1979), *Shortest–Route Methods: 1. Reaching, Pruning, and Buckets*, **Operations Research**, 27(1), 161–186.

Dial, R.B. (1969), *Algorithm 360: Shortest Path Forest with Topological Ordering*, **Commun. ACM**, 12(11), 632–633.

Dial, R.B., Glover, F. Karney, D. and Klingman, D. (1979), *A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees*, **Networks**, 9, 215–248.

Dijkstra, E.W. (1959), *A Note on Two Problems in Connection with Graphs*, **Numerish Mathematik**, 1, 269–271.

Divoky, J.J. (1987), *Improvements for the THRESH–X2 Shortest Path Algorithm*, **Operations Research Letters**, 6(5), 183–187.

Ford, L.R. (1956), *Network Flow Theory*, Report No. P–293, Rand Corporation.

Gallo, G, Pallottino, S., Ruggeri, C., and Storchi, G. (1982), *Shortest Path: A Bibliography*, Rapporto 27.82, CNR–PF, Informatica–SOFMAT.

Gallo, G. and Pallottino, S. (1986), *Shortest Path Methods: A Unifying Approach*, **Mathematical Programming Study**, 26, 38–64.

Gallo, G. and Pallottino, S. (1988), *Shortest Path Algorithms*, **Annals of Operations Research, Fortran Codes for Network Optimization**, 13, 3–79.

Gilsinn, J. and Witzgall, C. (1973), *A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees* , NBS Technical Note 772, US Dept. of Commerce.

Gondran, M. and Minoux, M. (1985), **Graphes et algorithmes**, Editions Eyrolles, Paris.

Glover, F., Glover, R. and Klingman, D. (1984), *Computational Study of an Improved Shortest Path Algorithm*, **Networks**, 14, 25–36.

Glover, F., Klingman, D. and Phillips, N. (1985A), *A New Polynomially Bounded Shortest Path Algorithm*, **Operations Research**, 33(1), 65–73.

Glover, F., Klingman, D., Phillips, N. and Schneider R.F. (1985B), *New Polynomial Shortest Path Algorithms and their Computational Attributes*, **Management Science**, 31(9), 1106–1128.

Hung, M.S. and Divoky, J.J. (1988), *A Computational Study of Efficient Shortest Path Algorithms*, **Computer and Operations Research**, 15(6), 567–576.

Kernigham, B.W. and Ritchie, D.M. (1988), *The State of C*, **BYTE**, McGraw–Hill publication, 13(8), 205–210.

Kernigham, B.W. and Ritchie, D.M. (1983), **Le langage C, Masson**, Paris.

Knuth, D.E. (1973), **The Art of Computer Programming. Vol. 1: Fundamentals Algorithms**, (Second Edition), Addison-Wesley, Reading, Mass.

Latour, D. (1987), *Une étude des méthodes de calcul des k meilleurs chemins*, Mémoire de maîtrise, Département d'informatique et de recherche opérationnelle, Université de Montréal, Direction S. Nguyen.

Lawler, E.L. (1976), **Combinatorial Optimization: Networks and Matroids**, Holt Rinehart and Winston, New-York.

Mondou, J.F. (1989), *Mise au point d'un prototype interactif–graphique pour la planification tactique du transport des marchandises*, Mémoire de maîtrise, Département d'informatique et de recherche opérationnelle, Université de Montréal.

Pallottino, S. (1979), *Adapatation de l'algorithme de d'Esopo-Pape pour la détermination de tous les chemins les plus courts: améliorations et simplifications*, publication #136, Centre de recherche sur les transports, Université de Montréal.

Pape, U. (1974), *Implementation and Efficiency of Moore's Algorithm for the Shortest Route Problem*, **Mathematical Programming**, *7, 212*.

Pape, U. (1980), *Algorithm 562: Shortest Path Lengths* **ACM Transactions on Mathematical Software** 6(3), 450.

Shier, D.R. and Witzgall, C. (1981), *Properties of Labeling Methods for Determining Shortest Path Trees*, **Journal of Research of the National Bureau of Standards**, Washington, 86(3), 317–330.

Tarjan, R.E. (1983), **Data Structures and Network Algorithms**, Society for Industrial and Applied Mathematics, Philadelphia.